Part 6

Code Generation

Let's Make Code

- A compiler ultimately has to make output
  - Assembly code
  - C code
  - Virtual machine instructions
- How do you do it?

Backing up....

- Let's enter a time machine and go <u>ALL</u> the way back to 4th grade math class
- Evaluate and <u>show your work</u>

```
2 + 3 * (10 - 2) + 5
```

Backing up....

- Let's enter a time machine and go <u>ALL</u> the way back to 4th grade math class
- Evaluate and <u>show your work</u>

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5
```

Backing up....

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5
```

Backing up....

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5
```

Backing up....

- Let's enter a time machine and go <u>ALL</u> the way back to 4th grade math class
- Evaluate and <u>show your work</u>

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5

31
```

Backing up....

- Let's enter a time machine and go ALL the way back to 4th grade math class
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

2 + 3 * 8 + 5

2 + 24 + 5

26 + 5

31
```

- This is exactly how a compiler does it!

Step-by-Step

- Imagine that you're only allowed to do one operation at a time (just like HW)
- Evaluate and <u>show your work</u>

```
2 + 3 * (10 - 2) + 5

t1 = 10 - 2     ; t1 = 8
```

- Perform each operation, put in a variable.

Step-by-Step

- Imagine that you're only allowed to do one operation at a time (just like HW)
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

t1 = 10 - 2      ; t1 = 8
t2 = 3 * t1      ; t2 = 3 * 8
```

- Perform each operation, put in a variable.

Step-by-Step

- Imagine that you're only allowed to do one operation at a time (just like HW)
- Evaluate and show your work

```
2 + 3 * (10 - 2) + 5

t1 = 10 - 2     ; t1 = 8
t2 = 3 * t1     ; t2 = 3 * 8
t3 = 2 + t2     ; t3 = 2 + 24
```

- Perform each operation, put in a variable.

Step-by-Step

- Imagine that you're only allowed to do one operation at a time (just like HW)
- Evaluate and <u>show your work</u>

```
2 + 3 * (10 - 2) + 5

t1 = 10 - 2     ; t1 = 8
t2 = 3 * t1     ; t2 = 3 * 8
t3 = 2 + t2     ; t3 = 2 + 24
t4 = t3 + 5     ; t4 = 26 + 5
```

- Perform each operation, put in a variable.

Control Flow

- Programming languages have control-flow

```
if a < b {
   statements
} else {
   statements
}

while a < b {
   statements
}
```

- Introduces branching to the underlying code

Basic Blocks

- Consecutive statements often appear in groups

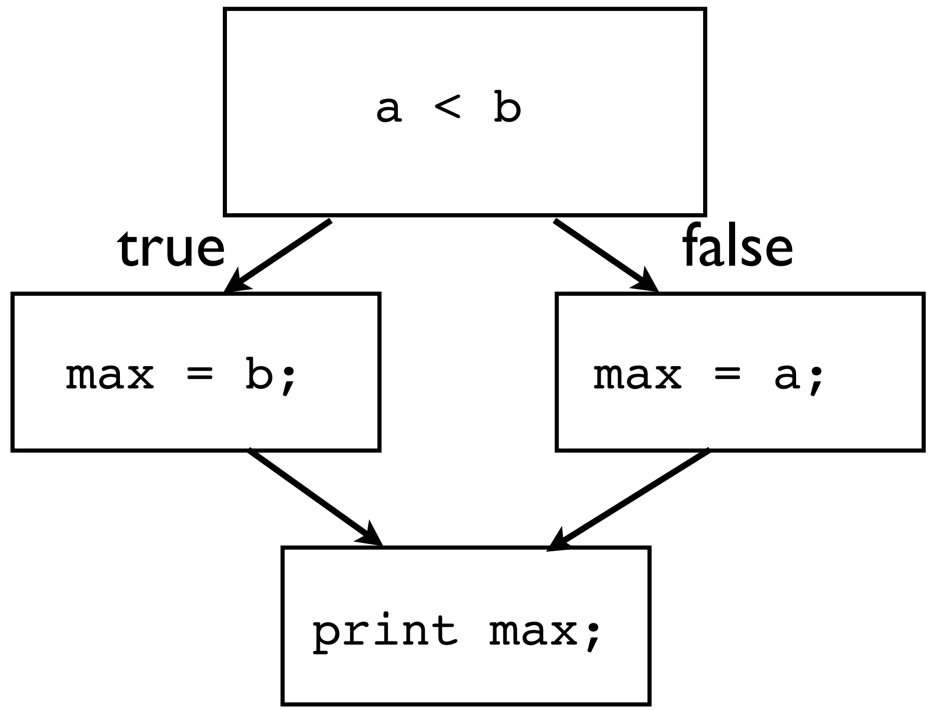
```
var a int = 2;
var b int = 3;
var c int = a + b;
print(2*c);
...
```

- A sequence of statements with <u>no change</u> in control-flow is known as a "basic block"

Control-Flow

- Control flow statements break code into basic blocks connected in a graph

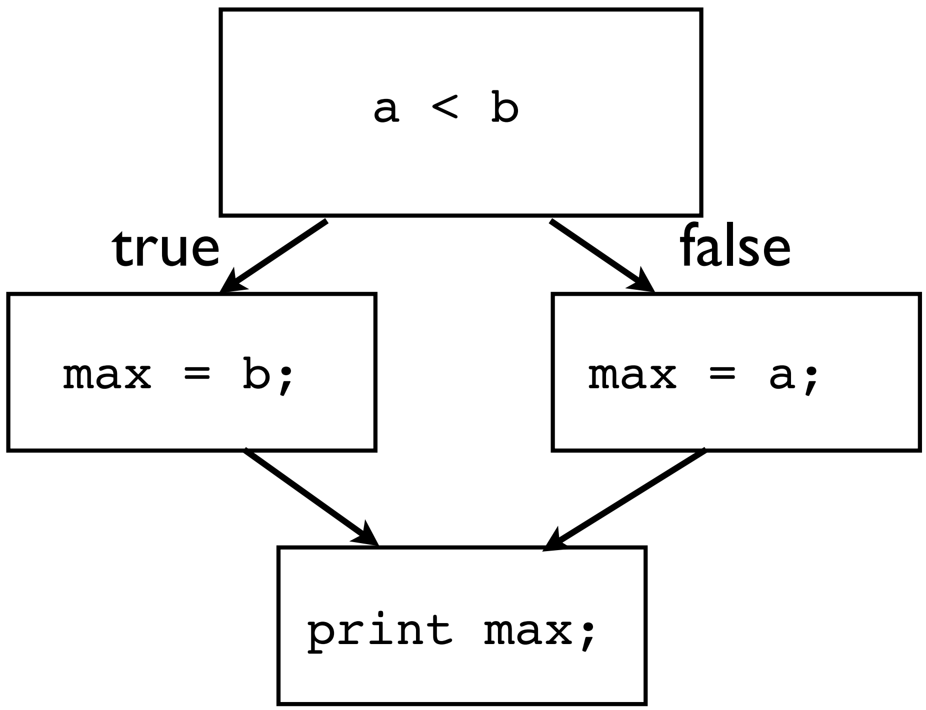
```
var a int = 2;
var b int = 3;
var max int;

if a < b {
   max = b;
} else {
   max = a;
}

print max;
```



- Control flow graph

Problem

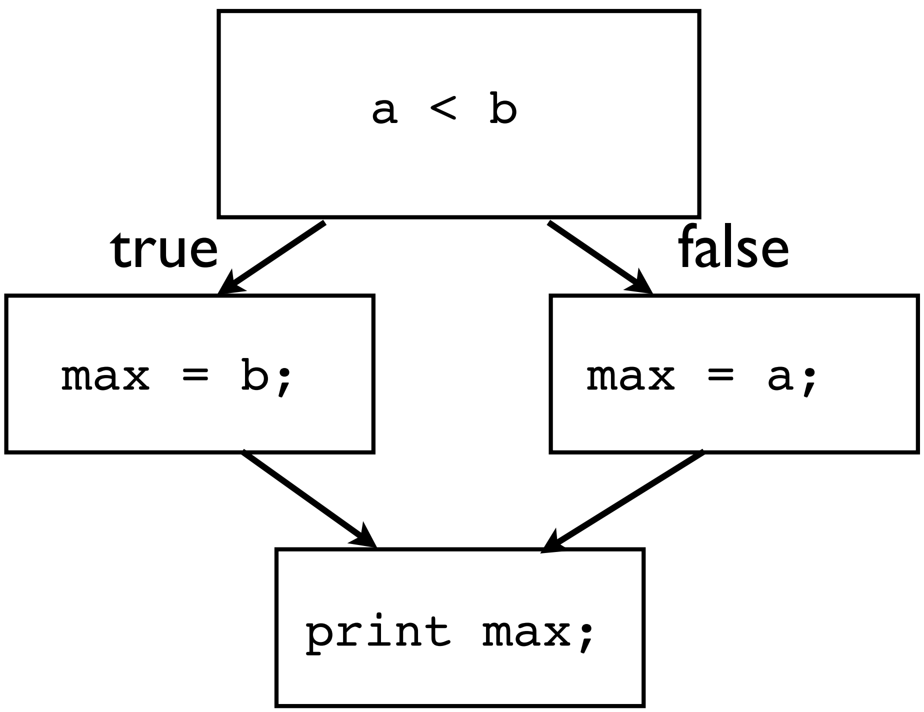
- How do you encode the control-flow graph into intermediate code?



- How is control-flow expressed?

One Approach: Gotos

- Label each block and emit jump/gotos




```
b1: test = a < b
    if (test) goto b2;
    goto b3;

b2: max = b;
    goto b4;

b3: max = a;
    goto b4;

b4: print max;
```

Implementation

- Code generator must emit unique block labels
- Blocks must be linked by goto instructions
- Visit all code branches

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
```

current block

...

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

current block

```
...
test = a < b
```

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

current block

```
...
test = a < b
```

Create labels

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

<u>current block</u>

```
...
test = a < b;
if (test) goto b2;
goto b3;
```

Emit gotos

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

Visit "true" branch

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

<u>current block</u>

```
...
test = a < b;
if (test) goto b2;
goto b3;

b2:
   statements1;
   goto b4;
```

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

Visit "false" branch

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
test = a < b;
if (test) goto b2;
goto b3;

b2:
   statements1;
   goto b4;

b3:
   statements2;
   goto b4;
```

Implementation Example

```
if a < b {
   statements1
} else {
   statements2
}
statements3
```

Start merge block

```
true_label = 'b2'
false_label = 'b3'
merge_label = 'b4'
```

current block

```
...
test = a < b;
if (test) goto b2;
goto b3;

b2:
   statements1;
   goto b4;

b3:
   statements2;
   goto b4;

b4:
   statements3;
   ...
```

Control-Flow Analysis

- There are many common programming errors related to control-flow issues
- Often a control-flow check is performed
- In addition to type checking.
- Will illustrate some common scenarios.

Dead Code

- There might be statements that never execute

```
while n > 0 {
    if n == 5 {
        break;
        print "Done!";    // <<<< Never executes
    }
    n = n - 1;
}
```

- Should it result in a compiler warning?

Uninitialized Variable

- What is the value?

```
var z int;
print z;
```

- Or this...

```
var z int;
if x > 0 {
    z = 10*x;     // Only initialized on one branch
}
print z;
```

Unused Variable

- What about this?

```
var x = 42;
var z = x + 10;   // z never reference ever again
...
<END>
```

- Does the compiler see the lack of use?
- Note: Such problems often the domain of linters/code checkers.

Project

- Turn Wabbit into C code
  - See `wabbit/c.py`
- Commentary: This might feel like "cheating", but a lot of compiler/language projects target C--especially in the early stages of development. C is both low-level and high-level enough to be useful for working out ideas, prototyping, debugging, etc.